Cooperative Caching of Dynamic Content on a Distributed Web Server

Vegard Holmedahl, Ben Smith, Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{veho, besmith, tyang}@cs.ucsb.edu

Abstract

In this technical report we propose a new method for improving the average response time of Web servers by cooperatively caching the results of requests for dynamic content. The work is motivated by our recent study of access logs from the Alexandria Digital Library server at UCSB, which demonstrates that approximately a 30 percent decrease in average response time could be achieved by caching dynamically generated content. We have developed a distributed Web server called Swala, in which the nodes cooperatively cache the results of CGI requests. We use a two-level cache table consistency protocol and a replicated global cache directory to maximize the system performance and minimize overhead in responding to dynamic Web requests. Our experiments show that the single-node performance of Swala without caching is comparable to the Netscape Enterprise server, that considerable speedups are obtained using caching, and that the cache hit rate is substantially higher with cooperative cache than with standalone cache.

1 Introduction

World Wide Web usage has experienced explosive growth in the last few years. In order to accommodate the growth, advances in server server technology are needed to improve the response time to the client under heavy load conditions. Typical methods for improving performance are file caching with proxies [6, 16, 7], and load balancing multi-node Web servers [2, 9, 14]. Research shows that for file fetches on the Web, the network is responsible for a significant portion of the response time. Web proxy caching is effective because it reduces the network bottleneck by keeping copies of files closer to clients. We take a different approach to improving Web access performance, recognizing that for some Web sites, processor utilization rather than network bandwidth is the bottleneck. This applies especially to sites making extensive use of requests for dynamic content, such as CGI requests. Our solution is a distributed Web server, called Swala, which cooperatively caches the results of CGI requests.

Our work is motivated, in part, by our experience with the Alexandria Digital Library (ADL) system [1] developed at UCSB. The current ADL system provides on-line browsing and processing of digitized maps and other geo-spatially mapped data through the Web. The activity of the ADL server includes computation and disk I/O for accessing compressed multi-resolution images with hierarchical data structures, as well as spatial database queries and on-the-fly HTML page generation. The limiting factor for this workload is the CPU rather than the network. Our study of the access logs of the ADL reveals that dynamic requests typically take at least an order of magnitude longer to complete than file fetches. Furthermore, we find that dynamic requests often are repeated, so caching the results of these requests would yield a considerable performance improvement. Our work is applicable to any Web site making extensive use of dynamic requests with long execution times, given a reasonable amount of repetition.

This technical report is organized as follows. Section 2 describes related work. Section 3 discusses our analysis of a recent access log from the ADL server. Section 4 presents the design of our system and consistency protocol. Section 5 presents the experimental results. Section 6 concludes the paper.

2 Related work

Numerous papers [6, 16, 7] discuss caching of Web content; however, they focus on caching static data and intentionally avoid caching dynamic data. Furthermore, they concentrate on proxy caching rather than server-side caching. Proxy caching of dynamic content may not be feasible because it does not permit caching of authenticated content and it hinders cache replacement algorithms based on job execution time.

Gwertzman and Seltzer [11] provide motivation for our research, by noting a tenfold increase in dynamically generated pages in six months, and suggesting that this increase in dynamic requests must be met by caching the scripts that generate the dynamic results. While their conclusions on the increasing amount of dynamic requests are interesting, their proposed approach has several disadvantages. First, it stops short of solving the problem of CPU contention: if a script takes several seconds to run, a client is not going to see improved response time just by having the script executed on a nearby proxy server rather than at the original server. Secondly, most dynamic scripts rely heavily on source files available on the original server. These would have to be transferred and cached at the proxy server before it could produce the same output as the original server. Thirdly, unless the script is platform-independent, the server and the proxy must be the same platform, for the proxy to execute the request. Finally, the proxy must be trusted by the original server to receive the scripts and source files. Most proxies operate as normal HTTP clients toward servers; this is not possible with Gwertzman and Seltzer's approach.

Caching the results of dynamic requests has been studied recently by IBM [13]. They have written a cache server and rewritten their server applications to insert and delete items in this cache. There are two main drawbacks to this approach. First, they require that the server application be rewritten to take advantage of the cache, and this can be a nontrivial task. Secondly, for every dynamic request, the Web server still must start the application, even if only to return a cache hit. For call mechanisms such as CGI, the operating system overhead for this call is significant, as we demonstrate in one of our experiments in Section 4. Our work overcomes both limitations, since our cache is built into the Web server.

Several papers [2, 9, 14] address load balancing within a group of Web servers to improve scalability. However, these papers do not study caching.

3 Access log analysis

In this section we illustrate the benefits of dynamic request results caching by analyzing an access log from the Alexandria Digital Library (ADL) at UCSB [1]. We have studied its log for September and October 1997, which contains a total of 69,990 requests. After filtering out HEAD and POST requests [5], we have re-sent the requests to the server and timed them. Illegal requests have been removed from the result file before analyzing the statistics, so the total number of requests studied is 69,337, of which 28,663 (41.3%) require execution of a CGI program. The longest running request among those studied is 127.7 seconds. We have found that the total service time for the 69,337 requests is 46,156 seconds, with an average response time of 0.666 seconds. The average response time for a file fetch is 0.03 seconds, and the average response time for a CGI is 1.6 seconds, giving two orders of magnitude difference between the execution of a CGI program and a file fetch. Moreover, the execution of the CGI programs constitutes 97% of the total service time for this set of requests, although the programs account for only 41.3% of the total number of requests. This motivates our further study of the log, since it implies that there is significant potential for a reduction in response time by optimizing CGI requests.

Our analysis results are summarized in Table 1. The first column shows the lower time threshold for requests included in the detailed study. The second column shows the number of requests taking longer than that threshold. The third column shows the total number of repeated requests. The fourth column shows the number of entries needed in the cache to exploit all repetition. The fifth column shows the potential time saving by fetching the repeated requests from cache. The sixth column shows the percentage of the total service time that could have been saved by CGI caching. For instance, by assuming that we could cache any CGI request taking more than 1.0 seconds to complete, we determined that we would need 189 cache entries to avoid all repeated execution, and that these 189 entries would result in 2,899 cache hits. By adding up the execution time for the requests that could have been cache hits, we found that 13,241 seconds could be saved. With a total service time of 46,156 seconds, this yields a potential reduction in average response time for this workload of almost 29 percent.

Time threshold	#long requests	Total # repeats	# unique repeats	Time saved (s)	Saved %
0.5 sec.	15153	7164	364	15957.3	34.6
1.0 sec.	7581	2899	189	13241.3	28.7
2.0 sec.	6721	2673	155	12840.4	27.8
3.0 sec.	6627	2666	151	12811.5	27.8

Table 1: Potential time saving by caching CGI.

Our analysis of different time thresholds illustrates one of the trade-offs in caching. If we cache too many short requests, we risk having a working set that exceeds our cache size, resulting in cache thrashing and no performance improvement. On the other hand, if we only cache very long requests, we will not realize as much of the benefit of caching. The threshold needs to be selected carefully, based on the system workload. More advanced cache replacement algorithms can alleviate some of the problem, by keeping the most important requests (in terms of execution time, access frequency, time of access, size etc.) in the cache. We discuss cache replacement algorithms in greater detail in section 4.3.

While our log analysis has been limited to a digital library, we expect that other Web sites that make extensive use of dynamic requests also can benefit from dynamic content caching. This is verified by recent work [13].

4 Design of the Swala distributed Web server

Swala is a multi-threaded, distributed Web server that runs on a cluster of workstations and shares cache information and cache data between nodes. The server saves the output from CGI [8] programs that are run as results of HTTP requests and stores information about the cached data in the cache directory. Each node communicates with the others to exchange cache data and meta-data. All execution is multi-threaded, to maximize throughput. Multi-threading raises consistency issues, which we discuss in section 4.2. We use memory-mapped I/O whenever possible to minimize the number of system calls and eliminate double-buffering. Multi-threading and memory-mapped I/O are important components of efficient Web servers [12].

4.1 Module design

As illustrated in Figure 1, every Swala node contains two primary runtime modules:

HTTP Module The control module starts and shuts down all other threads. After initial startup, it listens for HTTP requests from clients and starts a thread for each incoming request (hereafter called a *request thread*). The request thread is responsible for the request from parsing to completion.

Cacher Module The cache manager maintains the local cache directory (which contains information about all entries cached on all nodes), and provides information to request threads. It contains three daemon threads. The first thread receives information about cache insertions and deletions from the other nodes, and updates the local directory. The second thread listens for cache data requests from the other nodes and starts a separate thread for each request to return the cache contents. The third thread wakes up every few seconds and purges the cache if necessary. The design of the purge cache routine is described in detail in subsection 4.3.

The rest of this subsection is devoted to cache design issues.

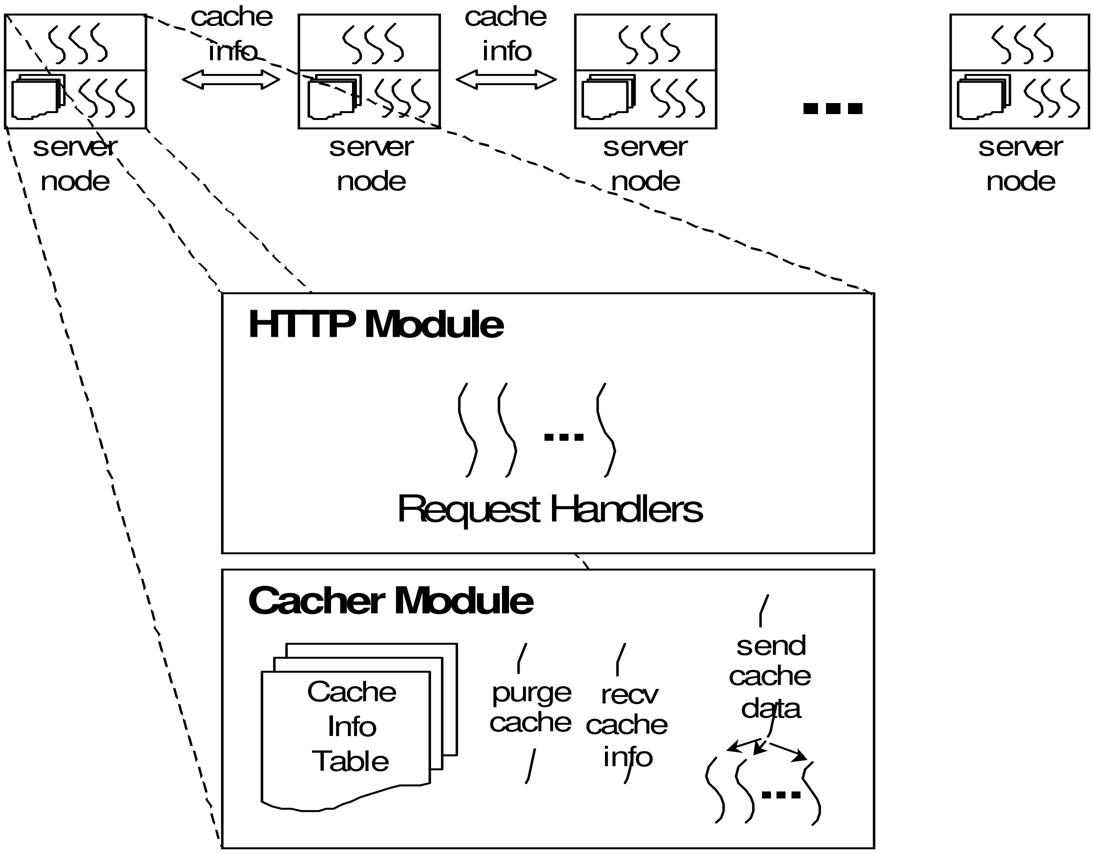

Figure 1: Components of the Swala architecture.

We have limited our cache to contain the results of CGI requests; files are not cached. There are two reasons for this. First, our access log analysis in Section 3 shows that execution of dynamic requests generally takes orders of magnitude more time than file fetches. Secondly, previous work on Web file caching [6, 16] has determined that for file requests, the network is the bottleneck, so file caching should occur as close to the client as possible, i.e. at a proxy server rather than at the base Web server.

We store only the cache directory in main memory, and use a separate operating system file to store the results of each cached request. Thus, every cache fetch in effect becomes a file fetch. There are two reasons why we have chosen this design. A disk cache is simple to maintain, and can contain orders of magnitude more cache data than a memory cache for the same price. Furthermore, our implementation runs on a UNIX platform, which does extensive file system caching, so we expect any recently used, reasonably-sized file to be available in memory.

Not all CGI requests can or should be cached. For example, CGI scripts that return different results for different users should not be cached. Swala uses a configuration file, loaded at startup, to provide the system administrator with a flexible way to control which requests are cacheable. Thus, on an incoming request, the request thread asks the cache manager to analyze it and determine whether it is (1) uncacheable, (2) cacheable but not cached, or (3) cached. The following explains the action of the request thread in each case:

Uncacheable An uncacheable request is executed without any more communication with the cache manager.

Cacheable If the cache manager determines that a request is cacheable but not yet cached, it prepares a file to receive the request results and hands the file handle to the request thread. The CGI is executed, and its results are written to the cachefile and returned to the user. The CGI return value is checked. If the execution has been successful and the execution time is longer than a runtime-defined limit, the request handler asks the cache manager to add the information to the directory and broadcast it to the other nodes in the group. Otherwise, the cache manager is asked to discard the results of the request.

Cached A cached request is fetched from the (local or remote) cache in which it resides, and returned to the client. Due to the cache consistency protocol discussed below, information about a remote entry is locally available very soon after the remote entry itself is accessible. The delay depends on the network latency and load within the cluster. After a cache fetch, the cache manager on the node that owns the item updates meta-data statistics, to facilitate cache replacement.

Figure 2 illustrates the control flow for a CGI request on the swala server.

4.2 Cache Consistency Issues

We need to address two types of consistency in our web server design: content consistency and cache table consistency. *Content consistency* is concerned with returning the most recent copy (or result) of a document or script. This issue is similar to cache invalidation in a shared memory system. In the context of caching dynamic content, strong content consistency requires that if the CGI is to execute again, the new result is identical to the cached result. Weaker consistency relaxes this requirement, allowing access to "old" data for a period of time. The second consistency issue is a result of the distributed, multi-threaded design of our system. We call this *cache table consistency*, and it is concerned with avoiding concurrent updates (which might cause data corruption), as well as maintaining consistent information in all nodes' cache dictionaries.

We implement a relatively weak consistency protocol for content consistency by allowing the system administrator to set a TTL (Time To Live; see Cache Replacement Policy section below) field for different CGIs. With this value set, the cached results will expire after a predetermined interval. Our scheme is reasonable for Web sites with infrequent updates of data and CGI scripts. This is true in digital library applications, where the material available through the Web server normally is read-only. We plan to investigate cache entry invalidation methods in future versions of Swala. This can be implemented by receiving invalidation messages from applications after the model of Iyengar et al. [13], and / or by monitoring the input of the CGI programs whose output is being cached, to detect invalidation, as suggested by Vahdat and Anderson [19]. We address cache table

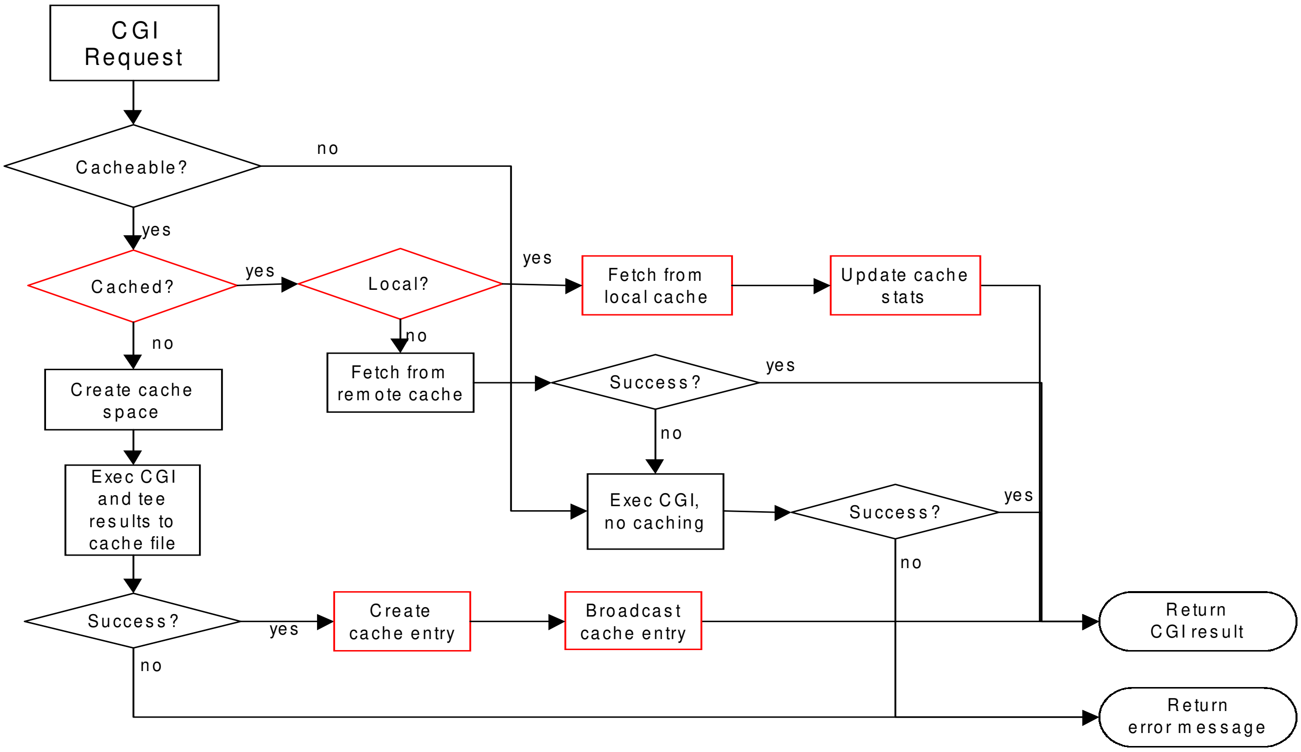

Figure 2: Swala control flow for handling CGI requests.

consistency with a two-level consistency protocol: intra-server and inter-server consistency. In the following section we describe our cache table consistency protocol.

Our intra-node consistency protocol protects the internal cache directory against simultaneous updates, avoiding potential data corruption. Every node contains a directory with one table for each node in the group, and each table contains information about what is cached at the corresponding node. There are three possible levels of locking that can be used to prevent data corruption in the cache directory: we could lock the whole directory for each access, lock only a table at a time, or lock each individual cache entry. (A fourth option is multi-granularity locking, for instance using entry locks on one table while using table lock on the other tables. We have not considered multi-granularity locking in this implementation.) We implement locking at the table level, with read- and write-locks to protect the cache table, in order to minimize lock contention while maximizing scalability. The process involved in satisfying a typical cacheable request motivates our choice of locking strategy: for every cacheable request received, we check the cache table for each node to see if any node has the request cached. If there were only one lock for the whole directory, we would experience unacceptable lock contention. If we used the third option, a typical cache lookup would involve a significant number of locks and unlocks, since the lookup would search a portion of each table in a directory. This would in turn seriously hamper scalability, since every added server would increase the number of locks & unlocks on lookup by the cache size.

With our inter-node consistency protocol, we are forced to compromise between strong consistency and minimal overhead. For instance, a strong inter-node protocol which ensures consistency between the cache directories on all nodes at all times would be quite expensive, since it would require a variation of a two-phase commit protocol between the nodes with synchronization on every cache update. Our choice of consistency protocol is motivated by a desire to maximize scalability and minimize critical path delay. This motivation has resulted in a relatively weak consistency protocol in the sense that cache updates are done asynchronously among the nodes without any global locks. More specifically, when a cache entry is inserted or deleted on a node, this information is broadcast to the other nodes, and they update their directories accordingly.

With the protocol described above, it is possible that a node may redo a cached request or attempt to fetch a deleted cache entry. We explain these situations below.

False cache miss We define it as a false cache miss when the node re-runs a CGI program instead of fetching the results from the cache. There are two situations when this could happen. If a node receives an uncached request, and an identical request arrives at the same node again before it has completed the first request, the node will redo the request, rather than wait for the cached results of the first request. The second situation where a false cache miss could occur is caused by our loose inter-node consistency protocol. If a node A caches a request, and another node B receives an identical request before it receives the cache update information from A, node B will redo the request instead of fetching the cached data from A. In this situation, the same information will be cached at two nodes. Both situations will occur rarely, since it is highly improbable that two identical requests will arrive within the relatively small time window that it takes to execute the CGI.

False cache hit We define it as a false cache hit when a node requests a cache entry that has been deleted from the cache of a remote node. This happens if node A asks node B for an entry after B deleted it and before A was aware of the deletion. The time interval in which this can happen is quite small, since the latency between the nodes is expected to be low. Furthermore, the cost of a cache miss on remote cache is small: when node A receives the cache miss response, it will execute the CGI request locally, as shown in figure 2. The client will only experience the added delay of a request/reply session between the two nodes, and this delay will in most cases be more than an order of magnitude smaller than the execution time of the request.

4.3 Cache Replacement

There are two aspects to cache replacement: when to initiate it, and what to replace. For the timing aspect, we have implemented two options. First, any insert that needs space can delete entries before inserting the new entry.

Secondly, the purge-cache background thread wakes up every few seconds and deletes entries from the cache if it deems that necessary:

1. If the current cache replacement model is Time-To-Live (defined below), items that have outlived their time must be purged. In this case, the thread calls a method to check the time left to live of every entry in the local cache, and delete stale entries.

2. If the current number of entries in the cache is over the high watermark for number of entries, the thread calls a method to delete the 'stalest' entries until the number comes below the corresponding low watermark. We currently use 98% and 90% as the high and low watermark, respectively.

3. If the current disk space usage in the cache directory is above the space high watermark, the thread calls a method to delete the 'stalest' entries until space is freed to below the low space watermark. We use the same values for the size watermarks: 98% and 90%.

Our reasoning behind the two different sets of watermarks is simple; many free cache entries does not necessarily mean that there is room for another file in the cache directory, and free space in the cache directory is of no use if the cache table in memory is full. By removing the 'stalest' entries whenever local cache usage exceeds any of the two limits, we minimize the necessity of deletion by request threads.

Once the need for cache replacement has been established, the cache replacement policy determines which entry or entries should be deleted. We have implemented Time-To-Live (TTL), where the file in the cache with the least time left to live is thrown out first, since this model gives a basic form of content consistency. If content consistency is deemed unnecessary (for static source files, as is the case in most digital library Web sites), four other replacement policies are implemented, and can be initiated at runtime:

Least-Recently-Used (LRU) The file that has the oldest access (usage) time is thrown out first.

Least-Frequently-Used (LFU) The file with the least number of accesses is thrown out first.

Execution time (TIM) The file with the shortest execution time is thrown out first. This is the equivalent of a size-based policy in a file cache attempting to minimize network contention.

Usage-Size-Execution time (USE) Each cache entry is given a USE-value on the basis of its frequency of use, size, and execution time. The entry with the lowest USE-value is thrown out first.

The 'best' replacement policy depends on the Web server's workload. We plan to investigate this area further.

5 Experiments

We have implemented and tested the Swala distributed Web server in a Unix environment, using TCP/IP communication between the nodes. Our testbed consists of six Sun 143-MHz Ultra 1 and two Sun Ultra 2 (dual 167-MHz CPU) workstations, with 64 or 128 MB of memory, connected by a fast (100 Mbit) Ethernet. To avoid the impact of using multiple processors during speedup experiments, we use only one CPU on the Ultra 2 nodes during the tests. Thus, the CPU power is roughly equivalent on all nodes. We present three sets of experiments. The first set compares the single-node performance of Swala with two popular Web servers and examines the overhead and benefit of caching on a single node. The second set of experiments concerns scalability; we examine the performance on an increasing number of nodes and study the benefit and overhead of caching in the clustered environment. The last set examines the cache hit rate in Swala and illustrates the advantage of cooperative caching over standalone caching. For all these experiments, clients are located within the campus network to avoid Internet bandwidth fluctuations.

5.1 Single-node performance of Swala and overhead of cache fetch

We present the results of a series of experiments comparing Swala with NCSA's HTTPd Version 1.5.1 [18] and Netscape Corporation's Enterprise Server [17]. The purpose of these experiments is twofold. We compare the efficiency of Swala with popular research and commercial servers, for a workload consisting of file fetches and CGI execution. We also measure the overhead of a local and remote cache fetch imposed by our cache system on a cached request. We use NCSA HTTPd because it is a widely used research package, and we use Netscape Enterprise because it is one of the most efficient commercial Web servers available [12]. We use WebStone [20], a standard Web benchmarking tool, to launch requests for the experiments in this sub-section.

# clients	HTTPd	Enterprise	Swala
6	0.207	0.032	0.058
12	0.410	0.068	0.073
18	0.604	0.105	0.099
24	0.838	0.140	0.133
30	1.146	0.174	0.164
36	1.304	0.212	0.199
42	1.709	0.248	0.235

Table 2: File fetch average response time in seconds measured using WebStone.

The first set of experiments is limited to file fetches: a 500 byte file is requested 35% of the time; a 5 Kb file is requested 50%; a 50Kb file is requested 14%; a 500Kb file is requested 0.9%, and a 1Mb file is requested 0.1% of the time. The average response times of Swala, Netscape and HTTPd are summarized in Table 2. The results show that the average response time of Swala is between 2 and 7 times faster than HTTPd. One reason for HTTPd's low performance is that it uses processes rather than threads. Enterprise is slightly faster than Swala on a small number of clients and slightly slower on a higher numbers of clients.

We further compare the performance of Netscape, HTTPd and Swala for CGI program execution. For these experiments, we use WebStone on three client machines, with 24 client processes sending the same request: a CGI program called `nullcgi` that does no work and produces less than a

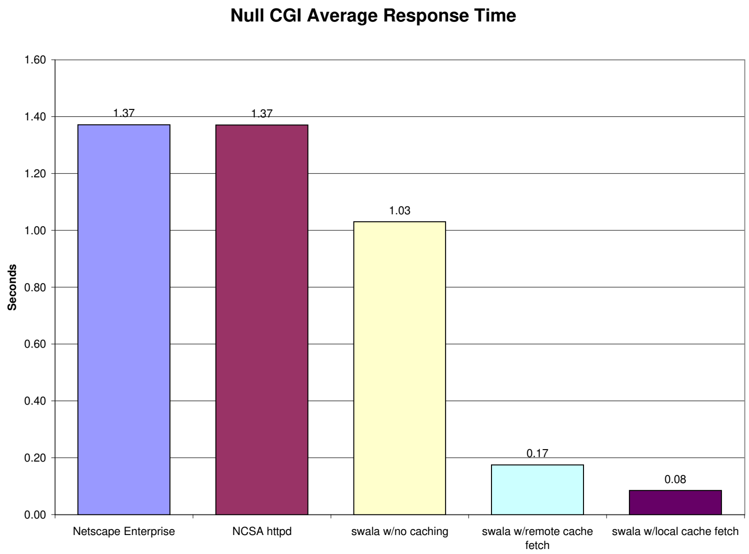

Figure 3: Overhead of Null CGI.

hundred bytes of output. This enables us to measure the overhead of CGI program execution, and compare it with the overhead of a cache fetch. For the Enterprise, HTTPd, Swala no-cache, and Swala local-fetch experiments, we run the Web server on a single node. For the Swala remote-fetch test, we start Swala on two nodes. The cache on the first node is initially warmed with the CGI request, and then all the requests from WebStone are sent to the second node, forcing it to fetch everything from the cache of the first node. This enables us to measure the cost of a remote cache fetch. The average response time for each Web server and configuration is depicted in Figure 3, which shows that Swala with cache disabled takes 25% less time than HTTPd and Enterprise. We therefore conclude that our CGI implementation is efficient. Note that 24 simultaneous clients and a fast network gives a considerable load on the server. These numbers can therefore not be read as the actual overhead per request; a more accurate number per request would be 1/24 of the times in the figure.

Figure 3 also shows that on a heavily loaded machine, it takes 0.17 seconds to fetch an entry from remote cache, and 0.08 seconds to fetch it from local cache. Thus, the overhead of remote cache fetch, even under heavy load, is only 0.09 seconds. Note that the extra cost of sending the request results to the client will be identical for local cache fetch and execution (remote cache fetch will add the overhead of fetch from one node to another), so the overhead described is relatively constant for any size output. Keeping in mind that the only time saved by caching these requests is the actual startup (fork, exec) of the CGI program, we can conclude that any dynamic request taking a reasonably long time (e.g. twice the time of a remote cache fetch) has the potential to benefit from the Swala caching mechanism.

5.2 Swala multi-node performance and cache insertion overhead

In this sub-section, we test the performance of Swala with an increasing number of nodes. We have created a synthetic workload by using the results of our analysis in Section 3, taking the CGI requests from the ADL access log and replacing each URI with a call to a synthetic CGI program that takes its own CPU-time and output size as parameters. For instance, the request

```
GET /ADL/Catalog/0 HTTP/1.0
```

with execution time 4.598 seconds and output size 8752 bytes is translated into

```
GET /cgi-bin/nph-workcgi.cacheable?4&ITER=4598&SIZE=8752 HTTP/1.0
```

An additional integer argument to the program ensures that two different requests in the original log remain different in the synthetic request file. Thus, without recreating the ADL in our lab, we have created a workload containing the same number of repeats and the same amount of temporal locality as the original log.

For the first experiment in this sub-section, two clients start eight threads each, and every thread launches requests to a single server node. Since we have sixteen client-threads, a varying number of server nodes, and request execution lengths that vary from one second to two minutes, the load on the nodes may not be evenly distributed. This setup gives a fairly accurate representation of a typical Web server cluster: clients ask a domain-name server (DNS) for the IP address of the Web server they want to access. The DNS answers with different addresses in round-robin fashion, so that the clients initially are distributed evenly among the available Web servers in the group [14]. However, the next time a client wants to access information at the Web site, it will reuse the address from the previous request, bypassing the round-robin distribution of the DNS. This imbalance can be further adjusted by using redirection, as addressed in the SWEB project [2]. Our goal is to measure the benefit of distributed caching without any help from load balancing.

Figure 4 shows the average response times of Swala with and without caching, where the number of server nodes varies from one to eight. The results show that Swala with cooperative caching enabled yields a much lower average response time than Swala with caching disabled. For instance, the decrease in response time with caching on eight nodes is about 25%. The results also show that the performance of Swala scales well when the number of nodes increases. With eight nodes, the average speedup is about nine.

We have also measured the overhead of cooperative cache insertion. We want to ensure that our handling of a cache miss followed by insertion of a new entry, broadcast of the cache information to the other nodes, and update of the replicated cache directory on all nodes is efficient enough not to destroy the advantage offered by cooperative caching. We have created a workload where each request is unique and cacheable. This forces every new request to generate a cache miss and a cache insert. We perform two different experiments to measure the overhead of cooperative cache maintenance. The first experiments measures the cost of local insert and broadcast, while the second experiment quantifies the cost of maintaining the global replicated cache directory. In the

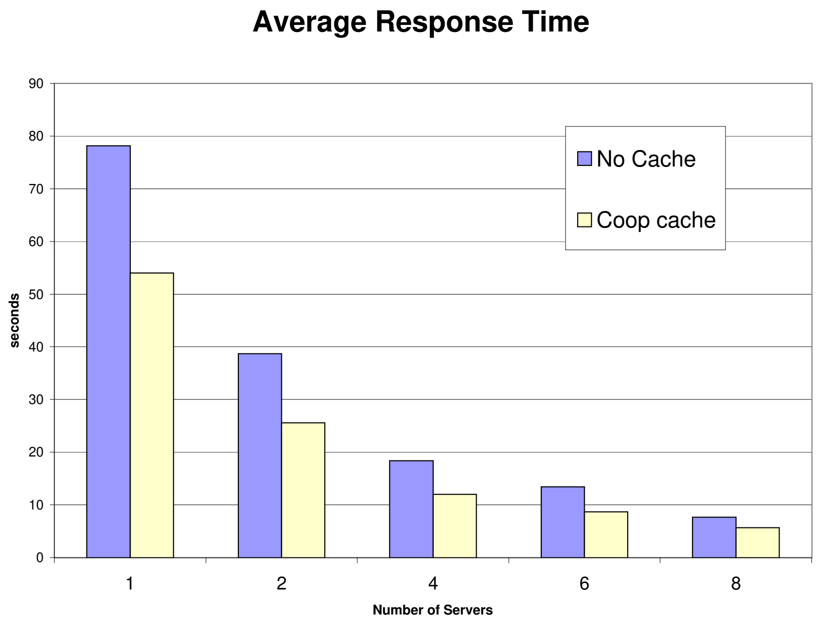

Figure 4: Multi-node performance of Swala with and without caching.

first experiment, we start Swala on two and eight servers, in non-caching and cooperative caching mode. We send 180 requests, each of which will run for one second on an unloaded CPU, to one of the nodes in the group, and the response time from this node is measured. Thus, the difference between non-caching and cooperative caching is that in the second case, Swala performs a cache miss, a cache insert, and broadcast to the other node/s. As Table 3 clearly shows, the overhead is insignificant and independent of the number of server nodes. We therefore conclude that the overhead of cache insert and cache information broadcast is negligible.

# nodes	No cache (s)	Cooper. Cache (s)	Increase (s)
2	0.965	0.992	0.027
4	0.963	0.995	0.032
6	0.961	0.994	0.033
8	0.967	0.993	0.026

Table 3: Response time overhead of cache insertion and cache information broadcast.

In preparation of the second overhead experiment, we have created a pseudo-server program which does not serve requests; it only sends cache directory updates to a Swala node. This enables us to simulate a complete 8-node Swala execution with minimal network disturbance: we start Swala on only one node, telling it that other nodes are running (to prepare its cache directory for cache information from eight nodes); we start the pseudo-server program to act as the other 7 nodes. The Swala node receives 180 uncacheable requests during the test, and each request takes approximately one second on an unloaded CPU. In our tests, cache information update messages are sent from the pseudo-server program, imposing the extra load of updating the local cache directory. The average response time for different levels of updates, with the base case as the first row, is reported in Table 4. The first column lists the number of updates per second (UPS) that the node receives in addition to its own work on uncacheable requests. The second column gives the average response time on requests for this rate of UPS, and the third column shows the increase in response time in seconds from the base case. As the table shows, the increase in response time on the one-second requests is insignificant.

UPS	Avg. Response time (s)	Increase (s)
0	0.958	0.000
10	0.970	0.012
20	0.976	0.018
30	0.976	0.018
40	0.990	0.032
50	1.005	0.047
60	1.018	0.060

Table 4: Response time overhead of replicated cache directory maintenance.

5.3 A comparison between cooperative and standalone caching

Our third set of experiments isolates and measures the advantage of sharing cache entries. We have used the analysis from Section 2 to determine the theoretical upper bound on cache hits for the requests issued: by counting the exact number of unique requests and repeats, we know how many cache hits are theoretically possible on a cache of infinite size. This upper bound on cache hits is compared with the actual number of cache hits on standalone caching and cooperative caching. The results are summarized in the two tables below. During each of these tests, 1600 requests are issued, 1122 of which are unique. In the cooperative tests, the nodes constituting the server are aware of each other and exchange cache information and cache data, while in the standalone test, each node caches what it receives and is unaware of any other node. Thus, a request may be cached several times in the standalone tests, whereas in the cooperative cache test, no cache entry is repeated (except for false cache misses). As the experiments show, the ability to reuse another node's cache entry rather than re-executing the script accounts for a large portion of the advantage of cooperative caching. The first column of tables 5 and 6 shows the number of nodes used. The second and third columns show the number of cache hits achieved for standalone and cooperative caching respectively. The fourth and fifth columns show how close to optimal caching this group of nodes comes, and the last column shows the improvement of cooperative caching over standalone caching.

The difference between the two tests is the cache size. In Table 5, the cache size is 2000, so that even a single node can cache every request, if necessary. Thus the only advantage that the cooperative cache has is that once *one* node has cached a request, no one else will have to execute it again (barring false cache misses). The results in Table 5 show that cooperative caching reaches a nearoptimal performance. Even though each individual node has sufficient cache space for all requests, cooperative caching still substantially outperforms standalone caching in terms of cache hit rate.

	# Cache hits		Percentage of Upper Bound		Cooperative vs. Standalone
# nodes	Standalone	Cooperative	Standalone %	Cooperative %	% improvement
1	466	N/A	97.5%	N/A	N/A
2	423	475	88.5%	99.4%	12.3%
4	383	466	80.1%	97.5%	21.7%
6	352	473	73.6%	99.0%	34.4%
8	327	473	68.4%	99.0%	44.6%

Table 5: Cache hit ratios for cooperative and standalone caching, cache size 2000.

Table 6 presents the results of a test with cache size 20. The consequence of the small cache size is that standalone caching (and cooperative caching with few nodes) is going to suffer considerably from cache overflow, having to continually replace entries in the cache. The table shows that even with a (cooperatively combined) cache size smaller than 14% of the unique cacheable requests received (on 8 nodes), the cooperative cache achieves over 70% of the upper bound of cache hits, while the standalone cache configuration achieves less than 40%.

	# Cache hits		Percentage of Upper Bound		Cooperative vs. Standalone
# nodes	Standalone	Cooperative	Standalone %	Cooperative %	% improvement
1	137	N/A	28.7%	N/A	N/A
2	140	208	29.3%	43.5%	48.6%
4	167	283	34.9%	59.2%	69.5%
6	177	320	37.0%	66.9%	80.8%
8	183	352	38.3%	73.6%	92.3%

Table 6: Cache hit ratios for cooperative and standalone caching, cache size 20.

6 Conclusions

In this paper, we have discussed a new approach to improving Web response time by cooperatively caching CGI results on a cluster of workstations. We have analyzed the access log from a digital library Web site and found significant potential for time saving through caching of dynamic content. Our experimental results demonstrate that the single-node performance of Swala is competitive with commonly used Web servers, the overhead of cooperative cache management is relatively small, and cooperative caching of CGI results substantially improves the average client response time.

Acknowledgements

We are grateful for valuable suggestions from Ambuj Singh and Anurag Acharya. Thanks to the members of the Parallel Processing Laboratory at UCSB for their patience during our intensive Web server tests. Thanks to Dan Andresen for collaborative research on early versions of the Web server (the SWEB project). This work was supported in part by the Norwegian State Educational Loan Fund, NSF IRI94-11330, NSF CCR-9702640, NSF CDA-9529418 and UC MICRO/SUN.

References

[1] D.Andresen, L.Carver, R.Dolin, C.Fischer, J.Frew, M.Goodchild, O.Ibarra, R.Kothuri, M.Larsgaard, B.Manjunath, D.Nebert, J.Simpson, T.Smith, T.Yang, Q.Zheng, "The WWW Prototype of the Alexandria Digital Library", *Proceedings of ISDL'95: International Symposium on Digital Libraries,* Japan August 22 - 25, 1995.

[2] D.Andresen, T.Yang, V.Holmedahl, O.Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers", *Proc. of 10th IEEE International Symp. on Parallel Processing (IPPS'96)*, pp. 850-856. April, 1996.

[3] D.Andresen, T.Yang, O.Egecioglu, O.H.Ibarra, T.R.Smith, "Scalability Issues for High Performance Digital Libraries on the World Wide Web", Proc. of the 3rd IEEE Forum on Research and Tech. Advances in Digital Libraries (ADL96), pp. 139-148, May, 1996.

[4] G.Bang, P.Druschel, "Measuring the Capacity of a Web Server", Proc. of the USENIX Symposium on Internet Technologies and Systems, December, 1997.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk, Hypertext Transfer Protocol – HTTP/1.0, RFC 1945, HTTP Working Group, May, 1996.

[6] P.Cao, S.Irani, "Cost-Aware WWW Proxy Caching Algorithms", Proc. of the USENIX Symposium on Internet Technologies and Systems, December, 1997.

[7] A.Chankhunthod, P.Danzig, C.Neerdaels, M.Schwartz and K.Worrell, "A Hierarchical Internet Object Cache", Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, March 1995.

[8] "The Common Gateway Interface", http://hoohoo.ncsa.uiuc.edu/cgi/.

[9] D.Dias, W.Kish, R.Mukherjee, R.Tewari, "A Scalable and Highly Available Web Server", Proc. of COMPCON 1996, Forty-First IEEE Computer Society International Conference: Technologies for the Information Superhighway, Santa Clara, California, February, 1996.

[10] S.Gadde, M.Rabinovich, J.Chase, "Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches", Workshop on Hot Topics in Operating Systems (HotOS), May 1997.

[11] James Gwertzman, Margo Seltzer, "World Wide Web Cache Consistency," Proceedings of the 1996 USENIX Technical Conference, San Diego, CA, Jan 1996.

[12] J.Hu, S.Mungee, D.Schmidt, "Techniques for Developing and Measuring High-performance Web Servers over ATM Networks," Washington University technical report #WUCS-97-09.

[13] A.Iyengar, J.Challenger, "Improving Web Server Performance by Caching Dynamic Data", Proc. of the USENIX Symposium on Internet Technologies and Systems, December, 1997.

[14] E.D. Katz, M. Butler, R. McGrath, A Scalable HTTP Server: the NCSA Prototype, *Computer Networks and ISDN Systems.* vol. 27, 1994, pp. 155-164.

[15] Paolo Lorenzetti, Luigi Rizzo, Lorenzo Vicisano, "Replacement policies for a proxy cache," draft, http://www.iet.unipi.it/ luigi/caching.ps.gz.

[16] R.McGrath, "Caching for Large Scale Systems", D-Lib Magazine, January 1996.

[17] "Netscape Server Central Index Page", http://home.netscape.com/comprod/server_central/.

[18] "The NCSA HTTPd Home Page", http://hoohoo.ncsa.uiuc.edu/.

[19] A. Vahdat, T. Anderson, "Transparent Result Caching," unpublished.

[20] "WebStone", http://www.sgi.com/Products/WebFORCE/WebStone/index.html.